Techniques for delayed binding of monitoring mechanisms to application-specific instrumentation points

Jeffrey Vetter*
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, USA, 61801

Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, USA, 30332

Abstract

Online interaction with computer systems and applications allows developers to monitor, experiment with, and debug long-running, resource-intensive applications at runtime. Traditionally, developers statically bind a monitoring mechanism to each application-specific instrumentation point. This approach has shortcomings for online, interactive monitoring. Namely, static binding limits portability among monitoring systems; it may mismatch monitoring mechanisms to interactive requests for monitoring data; and, predictions for the performance and execution paths of instrumentation for static bindings are left to the developer. To address these concerns, we have created a new technique called **monitoring assertions** *that allows monitoring systems to delay binding of monitoring mechanisms to application-specific instrumentation points until runtime. Our empirical results show that we can alter the performance of both the application and the monitoring system by removing static binding requirement of application-specific monitoring systems.*

1. Interactive monitoring systems

Interactive monitoring systems permit users to interact with the monitoring system during the execution of the target application. Users can enable, disable, and reconfigure the monitoring system to suit their exploration of the application [19, 17, 9, 6]. These interactive monitoring systems complement other forms of monitoring that use post-mortem analysis [25, 18, 16, 2, 3], but they share with such systems the need to analyze and display monitoring data [20, 11, 12, 4]. The interactive monitoring systems investigated by our group are those that permit end-users to capture *application-specific* information from targets, such as the position and velocity of molecules, or values of a solution matrix in a multi-grid solver. Some online monitoring systems, which include Paradyn, Quartz, and EEL [17, 2, 3], use modified compilers or executable editing to interrogate the application for performance-relevant information. However, we do not employ these techniques because they are limited in the types of information they can provide to the user [19], especially when executables are highly-optimized [7, 24]. Alternatively, users of application-specific monitoring systems annotate their application source code to supplement information available from these other monitoring methods. Our particular form of application-specific monitoring, called *event-based* monitoring [13, 4, 14], produces streams of events that represent observed application state; the user analyzes these streams judge target system behavior.

Motivation. Application instrumentation is a primary component of any application-specific monitoring system. Across a wide variety of systems [23, 9], instrumentation support may be described as follows. (1) The developer chooses basic instrumentation mechanisms and annotates the source code. (2) At runtime, the monitoring system receives data from this instrumentation. (3) The monitoring server has limited runtime control over the instrumentation such that it typically relies on instrumentation placement to produce interesting data at appropriate rates. This *static binding* of monitoring mechanisms to instrumentation points is unacceptable for interactive monitoring systems because it is not easy to reasonably predict the interactor's exploration scenario when deciding upon instrumentation mechanisms. The developer must also predict the interactor's exploration scenario to help maintain consistent (or meaningful) data views of the target application.

This paper introduces **monitoring assertions** that allow developers to instrument applications without forcing them to predict how monitoring data might later be used and without forcing them to judge *a-priori* which mechanisms

*NASA financially supported Vetter with a Graduate Student Researchers Program Fellowship while he was a Ph.D. candidate at Georgia Tech. This work was also funded, in part, by NSF equipment grants CDA-9501637, CDA-9422033, and ECS-9411846.

are most appropriate. These assertions annotate *what* application data is available and *when* it is accessible to the monitoring system.

Related work. *Monitoring assertions* are code annotations; we propose these assertions could be used by any monitoring system in place of their application-specific instrumentation. Then, the monitoring system would query and control the instrumentation using the 'monitoring assertions' library API. Our earlier annotated bibliography provides many references to these instrumentation systems [10].

Tools like gprof[8] and Quartz[2] provide the user with valuable information about execution frequency; they assign elapsed time to static, syntactic units, such as procedures or statements. Additional work by Ammons and colleagues [1] extends this work to use flow and context information for performance counters. Paradyn [17] provides similar control-based application performance information; it delays the insertion of instrumentation until requested by the interactor, whether it is a query driven by a human or by Paradyn's performance consultant. These systems [8, 2, 1, 17] rely on information stored within the executable by the compiler—either the instrumentation itself or instrumentation targets such as procedure entry points. In many cases, this limitation prohibits the use of executable editing to observe application-specific data. An example of this shortcoming is described in §3.

Application-specific monitoring systems such as Falcon[9] and Autopilot [21] provide their own instrumentation libraries and instrumentation controls. Developers insert this instrumentation into their application source code and then, capture and analyze data produced by the instrumentation at runtime. Unfortunately, most every application-specific monitoring system has its own nonportable instrumentation library. In this respect, monitoring assertions could provide a common instrumentation tool for both application developers and tool developers. While these earlier instrumentation techniques propose various instrumentation mechanisms [19, 5, 16, 23], they focus primarily on the design of the monitoring system itself.

Research contributions. The novel idea of this work is to generalize instrumentation so that the binding of specific monitoring mechanisms to instrumentation points in target applications is delayed until more is known about the expected use of the monitoring information (i.e., at runtime). Note that executable editing allows the monitoring system to delay binding until runtime; however, it then can only bind to a limited scope of application attributes, such as procedure entry points. Whereas, monitoring assertions allows developers to introduce as many application-specific binding points as necessary. (1) **Monitoring assertions** capture the desired properties of application-specific instrumentation of *what data to monitor* and *when to monitor that data* without the requirement of static binding to particular monitoring mechanisms. (2) **Instrumentation signatures** result in our system's ability to predict monitoring overheads and thereby enable runtime choices in instrumentation bindings based on observed patterns in instrumentation flow and frequencies. These signatures are derived from reference executions of instrumented applications. (3) Empirical evaluations of monitoring assertions and instrumentation signatures demonstrate the utility of delaying mechanism binding until runtime.

Sample applications. Two applications demonstrate our ideas. *Heat diffusion* is a 27-point, 3D-stencil, time-stepped simulation [22] implemented with kernel-level threads for SMP platforms. This simulation exhibits nearest neighbor sharing common to many simulations of physical systems. A more complex example is provided by the Splash 2 *OCEAN* benchmark. The Ocean benchmark simulates eddy and boundary currents in a cuboidal ocean basin. The application uses finite differencing CFD with a regular grid. The algorithm uses a red-black Gauss-Seidel multi-grid equation solver; each time-step of the simulation involves setting up and solving a set of spatial partial differential equations.

Experiment platform. The experimental platform is a network of 4 two-processor Sun Ultra 2 Model 2148s (148 MHz UltraSPARC CPU). Each system has 128MB of main memory and they run Solaris 2.5.1. Although this work focuses on shared memory architectures, we are not aware of any fundamental design restrictions such that monitoring assertions could not work in a message passing environment.

Paper outline. §2 reviews application instrumentation including various mechanisms and characteristics. §3 and §4 explain the technique of monitoring assertions and instrumentation signatures, respectively. §5 reveals our concluding thoughts and some future research issues.

2. Background: application instrumentation

Typical online monitoring systems have four basic components as illustrated in Fig. 1: system software, application, monitoring server, and interactor. Systems that broadly fit this model include Paradyn [17], Falcon [9], Magellan [22], Avatar [20], Vista [23], and various debuggers. We assume no special operating system, special compilers, or hardware. The interactor controls the server; it specifically provides user interface and visualization capabilities. The server is generally a separate process or thread that communicates with the application via shared memory or IPC.

Application-specific instrumentation is represented as software statements added to the target application [19, 5, 16]. This instrumentation gathers data about the application's execution, packages it, and sends it to some higher-level monitoring system. This instrumentation has two basic objectives: (1) instrumentation identifies *which* application

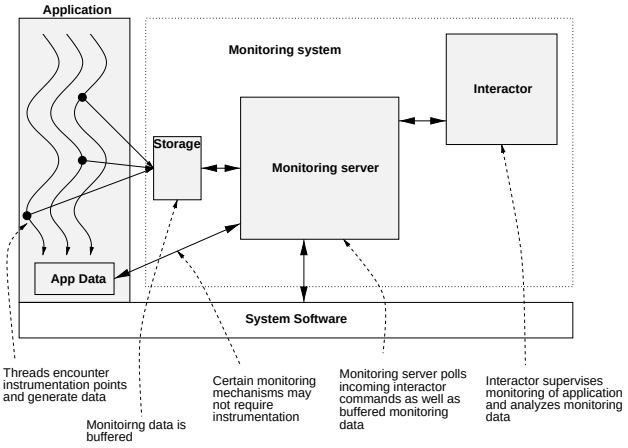

Figure 1. Typical model for an interactive monitoring system.

data the monitoring system can observe; and, (2) instrumentation identifies *when* the monitoring server can safely observe application data; thereby addressing both the validity of the each observed data item as well as the consistency of multiple data items.

The following code illustrates application-specific monitoring instrumentation for the Magellan computational steering system, which supports interactive monitoring. This code is taken from the Splash 2 Ocean benchmark.

```
AS_RegisterObject(steer, "residual norm",
  &multi->err_multi, AST_DOUBLE, 0, &sResNorm);
AS_Sense(sResNorm);
```

During initialization, the Magellan instrumentation registers target data using the call `AS_RegisterObject` and providing the name, type, size, and address to the runtime Magellan server. Next, a sensor is placed in the loop using `AS_Sense` to provide controlled access to these registered objects. Magellan need only enable these `AS_Sense` instrumentation points to receive periodic updates for this application variable.

To reason more specifically about instrumentation points, we now introduce some terminology. A **static instrumentation point** is uniquely identified by its location within the source code: a filename and line number. $IP_{file,line}$ describes this static instrumentation point. For convenience, we collapse $IP_{file,line}$ into a unique identifier: sid.

At runtime, however, this static description does not provide enough information to the monitoring system. In one scenario, multiple threads can execute the same instrumentation point, so the instrumentation point must also provide a thread (or process) identifier tid and context (or stack frame) information–sf–for scoping and recursion. Without additional compiler support, we currently do not explore use of sf to extend sid. Adding tid, a **runtime instrumentation point (rid)** is uniquely identified by sid and tid; or, $f(sid, tid) \rightarrow rid$.

In addition to the information generated by instrumentation points, the monitoring system may also accumulate and observe information about the instrumentation itself. The monitoring system tracks the number of times instrumentation points are encountered by application threads as well as the number of times data items are actually observed by each instrumentation point. A **hit** occurs when an instrumentation point is encountered by an application thread. $Hits(sid)$ gives the number of total hits on sid by any thread while $Hits(sid, tid)$ gives the number of total hits on sid by thread tid. For example, every time any thread executes the `AS_Sense` instrumentation point, it counts as one *hit*. A **capture** is a hit where the instrumentation point observes application state and forwards it to the monitoring system. $Captures(sid)$ gives the number of total captures on sid by any thread while $Captures(sid, tid)$ gives the number of total captures on sid by thread tid. All captures are necessarily hits; however, hits are not necessarily captures.

An **event** ϵ_{eid} is a quad-tuple containing an event class descriptor ($class$), a timestamp (TS), a location (loc) and state (S): $\epsilon_{eid} = (class, TS, loc, S)$. A unique event identifier (eid) identifies each event. $Timestamp(\epsilon_{eid})$ furnishes the timestamp of the event ϵ_{eid}. The instrumentation point generates the event timestamp when it creates the event. $Loc(\epsilon_{eid})$ furnishes the originating location of ϵ_{eid} which includes the tid as well as sid. $State(\epsilon_{eid})$ allows the monitoring system to access the event's state. State sizes vary for different event classes, so $Size(\epsilon_{eid})$ furnishes the state size of the event ϵ_{eid}.

2.1. Instrumentation mechanisms

We consider four types of instrumentation mechanisms: tracing sensors, sampling sensors, snapshot sensors, and synchronous probes. For these definitions, assume that each of the mechanisms are enabled and that rid has $Hit(rid) = n$.

A *tracing sensor* installed at rid captures n events ϵ and forwards each event to the monitoring server via a FIFO shared buffer. $Captures(rid) = Hits(rid) = n$. Fig. 2(a) furnishes the pseudo code for a tracing sensor instrumentation point. All events generated by a tracing sensor are consumed by the monitoring server. The application thread must contend with the monitoring server for exclusive access to the FIFO buffer. If the non-blocking `buffer insert` fails, then the application must retry the insert. The `buffer insert` could fail because either the FIFO is currently locked or it is full.

A *sampling sensor* captures $(Hits(sid)/\omega_{sampling})$

```
trace-sensor ( datahandle )
  hits++
  if enabled
  captures++
  allocate event
  copy datahandle to event
  package event
  while buffer insert fails
    buffer insert event
  return
```

(a) tracing

```
sampling-sensor ( datahandle )
  hits++
  if enabled
    if (hit % samplfreq) == 0
      captures++
      allocate event
      copy datahandle to event
      package event
      while buffer insert fails
        buffer insert event
  return
```

(b) sampling

```
snapshot-sensor ( datahandle )
  hits++
  if enabled
    captures++
    lock shared area
    copy datahandle to shared area
    update shared area attributes
    unlock shared area
  return
```

(c) snapshot sensor

```
synch-probe ( datahandle )
  hits++
  if enabled
    captures++
    mondata = datahandle
    barrier (monitor thread)
    barrier (monitor thread)
  return
```

(d) synchronous probe

Figure 2. Monitoring mechanism pseudo-code.

events and forwards these events to the monitoring server via a FIFO shared buffer. With $\omega_{sampling} > 1$, $Captures(rid) \leq \frac{Hits(rid)}{\omega_{sampling}}$. A tracing sensor is a sampling sensor where $\omega_{sampling} = 1$. Fig. 2(b) outlines the operation for a sampling sensor. $\omega_{sampling}$ can change dynamically.

Fig. 2(c) defines the operation of a *snapshot sensor* that captures n events but stores only one event in a shared location instead of placing it in a FIFO buffer. The application instrumentation overwrites this stored event with the most recent event. The monitoring server then reads this shared location to find this event.

A *probe* directly reads application memory without synchronizing with the application through an instrumentation point. Original probes[19] have no corresponding rid. The monitoring system copies the data immediately upon request. This lack of synchronization limits probe state size to machine-dependent data sizes. Due to this limitation, we do not study normal probes in detail; however, we do introduce synchronous probes as an alternative.

A *synchronous probe* synchronizes execution of the monitoring server and the application thread at rid. Once the synchronization occurs, the monitoring server copies ϵ_{eid} directly from the stalled application. No data is buffered between the application and the monitoring server. Fig. 2(d) illustrates the primary differences in operation of a synchronous probe from the previous constructs.

2.2. Instrumentation characteristics

Each of these instrumentation mechanisms have quantifiable impacts on the application and the monitoring system including application perturbation, monitoring latency, potential monitoring frequency, and the consistency of the monitoring information. We define γ as a global clock available to the application, instrumentation points, the monitoring server, and the interactor.

Each rid has a time associated with its execution. *Instrumentation point execution time*, $\Delta_{Execution}(rid) = \gamma_{stop} - \gamma_{start}$, is the difference in time from the call to the return of the instrumentation point rid. This $\Delta_{Execution}(rid)$ captures the time the instrumentation waits on mutex locks, full buffers or other costs.

Small changes in application instruction sequences from this instrumentation can result in substantial aggregate performance perturbation[15]. We measure this aggregate perturbation instead of the execution time of each individual instrumentation point. *Application runtime*, $\tau_{application} = \gamma_{stop}^{application} - \gamma_{start}^{application}$, is the runtime of the original application without any instrumentation points. Instrumented application runtime, $\tau_{instrumented} = \gamma_{stop}^{instrumented} - \gamma_{start}^{instrumented}$, is the runtime of the instrumented application. *Application perturbation*, $\Delta_{Perturbation} = \tau_{instrumented} - \tau_{application}$, is the difference in the runtime of the instrumented application minus the runtime of the original application. *Application perturbation slowdown* is the ratio of instrumented runtime to original application runtime: $\frac{\tau_{instrumented}}{\tau_{application}}$. $\Delta_{Perturbation}$ is a function instrumentation mechanisms used throughout the exploration of the target application. It is our experience that aggregate $\Delta_{Perturbation} > 0$ and perturbation slowdown is greater than 1. For our evaluation, we lock the instrumentation points to one mechanism for each benchmark's duration.

Tab. 1 details these measurements for two example applications. Each application used two processors. *Optimized* mode used compiler option `-O`; *debug* mode using compiler option `-g`; *gprof* mode used option `-xpg -O` on the SUNWspro compiler. *Optimized/ma* mode is the performance of the optimized application with additional monitoring assertions instrumentation. As we expected, debug mode increased the runtimes of each application considerably. Gprof has a aggregate perturbation of 28-33%. Monitoring assertions produced on 1-3% aggregate perturbation on the target applications. **Total hits** represents the cumulative sum of all assertion hits during execution of the application. We certainly expect a more thoroughly instrumented version of either application to produce more perturbation;

Application	State	Avg Runtime (sec)	Total hits	Ratio
OCEAN	Opt	336	-	1
	Opt/ma	337	3110	1.01
	Opt/gprof	430	-	1.28
	Debug	697	-	2.07
Stencil	Opt	124	-	1
	Opt/ma	128	202	1.03
	Opt/gprof	165	-	1.33
	Debug	252	-	2.03

Table 1. Application performance under different instrumentation scenarios.

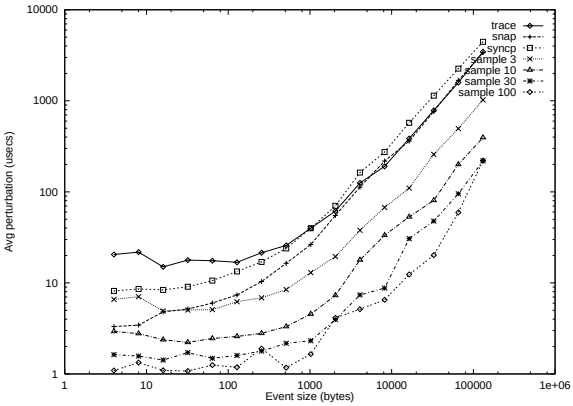

Figure 3. Average perturbation per instrumentation point versus event size.

however, this analysis demonstrates that the perturbation from monitoring assertions is not inherently limiting.

Online monitoring systems obviously require some latency to capture, transport, and analyze their data. This latency is a function of many factors including instrumentation mechanisms. Here, we are primarily interested in the latency between the event generation by application instrumentation and event consumption by the monitoring server. Buffer management can also alter the latency of the system[9, 23]; we hold buffer parameters constant across our evaluations.

Event latency, $\Delta^{eid}_{Latency} = \gamma - TS(\epsilon_{eid})$, is the difference of the current time from the timestamp on the event ϵ_{eid} as observed by the monitoring server. Assuming a correct globally consistent clock with infinite resolution, $\gamma > TS(\epsilon_{eid})$ and $\Delta^{eid}_{Latency} > 0$.

2.3. Perturbation and latency evaluation

Each of our earlier defined mechanisms have different perturbation and latency characteristics. To measure these characteristics, we have created micro-benchmarks and then, monitored those micro-benchmarks with the Magellan system[22]. As the code below illustrates, the benchmark is a loop that accesses `array` with size `datasize`.

```
int array[datasize]
start timer
for ITER
do
  incr all elements of array
  Instrumentation point(array)
  delay
done
stop timer
```

Perturbation is measured by subtracting the wall-clock time of the uninstrumented loop from the wall-clock time of the instrumented loop. The monitoring server calculates *latency* with each ϵ_{eid} timestamp and the same machine clock. This clock's resolution is microseconds.

Figs. 3 and 4 illustrate the clear differences in the perturbation and latency characteristics of tracing sensors, sampling sensors ($\omega_{sampling} = 30$), snapshot sensors, and synchronous probes. Characteristics such as buffer properties and interference have been studied by Waheed and associates, and Gu and colleagues [23, 9]. We recognize these issues by assuming a reasonable buffer size and a steady-state computer system.

From this analysis, we see that all perturbation increases with data size. Tracing sensors impose the highest perturbation on the application while sampling sensors have the lowest overall perturbation. However, this sampling sensor perturbation varies with $\omega_{sampling}$. Event latency for tracing and sampling sensors is almost identical regardless of data size because the underlying data structures are similar. the latency for synchronous probes and snapshot sensors is very predictable, increases with data size, and is almost two orders of magnitude better than either tracing or sampling sensors.

Given these clear differences, it is best to match the appropriate mechanism to current interactive monitoring requests. Our monitoring assertions allow dynamic binding of these mechanisms to the appropriate instrumentation points for each monitoring request. The monitoring system can choose which mechanism to employ.

3. Monitoring assertions

As in the earlier Magellan example, the instrumentation mechanisms are bound at compile time. `AS_Sense` dictates that the mechanism used to monitor `sResNorm`, etc. is always a sensor. If the interactive monitoring system needs a different type of mechanism such as a synchronous probe to minimize latency, then the user would have to re-instrument the application and recompile.

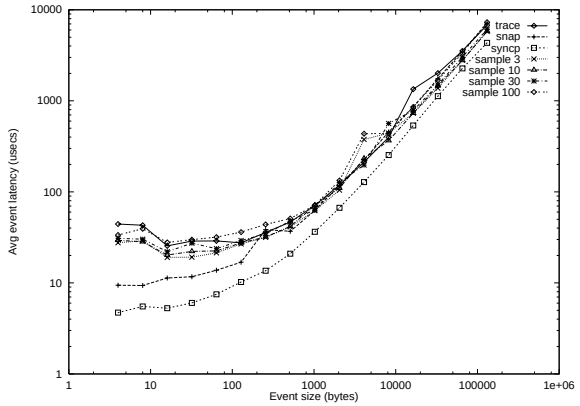

Figure 4. Average latency per event versus event size.

Monitoring assertions help solve this problem by providing the monitoring system with a method for dynamically binding these mechanisms to the assertions. Monitoring assertions allow the developer to generally instrument their application by identifying *which* data is available for monitoring and *when* this data is observable. They are annotations that a developer adds to source code. So, as a result of the scenario depicted in the Magellan example, that code becomes Fig. 5, where mechanisms are replaced by assertions. Developers can also limit the types of mechanisms that can be used at individual assertions that further limit the choices available to the monitoring system when selecting mechanisms.

Given our earlier analysis of different monitoring mechanisms, the following scenario illuminates the advantages of monitoring assertions over previously-discussed monitoring instrumentation. In Fig.5, the monitoring system can make different choices about which monitoring mechanisms to use at `AAA_Access` based on interactive requests.

If the user needs to track a variable such as `resNorm` with size 8 bytes, then the monitoring system could easily choose a snapshot sensor because it provides the best perturbation (from Fig. 3) with the second best latency of 7 microseconds (see Fig. 4). Synchronous probes provide the best latency at 8 bytes; however, the latency improvement is very small and synchronous probe perturbation is higher.

If the user wants to display a larger data item or additional data items, then the monitoring system must reevaluate its instrumentation choices. If the user requests the variable `f` with size 4112, the monitoring system can choose sampling sensors ($\omega_{sampling} = 30$) with the best perturbation. At large data sizes, the data copying dominates the latency costs; therefore, all mechanism latencies converge toward a common latency cost. This fact effectively reduces the monitoring system's choice dimensions to perturbation, which favors sampling sensors for this example.

```
AAA_double(multi->err_multi); /* resNorm */
AAA_int(k);    /* level */
while ((!flag1) && (!flag2)) {
  /* ...code omitted... */
  AAA_Access(multi->err_multi,wu,iter,
  k,f,tolerance);}
```

Figure 5. Splash2 Ocean with monitoring assertions.

Without monitoring assertions, the monitoring system would be unable to combine interactive monitoring requests with its knowledge about application instrumentation to provide these optimized solutions.

Localized application de-optimization. Most large-scale, high performance applications have thousands of data items of which only a small portion are useful for monitoring. Using application-specific knowledge, the developer chooses which data to make available and when. By placing the monitoring assertions in the code, the developer also chooses how to localize de-optimizations. Namely, in contrast to de-optimizations performed when using general debug options (e.g., `-g`) provided by compilers, monitoring assertions result in de-optimizations only for the specific code components where they are placed. Monitoring assertions can be easily disabled with command line switches or preprocessor directives.

The developer controls the tradeoff between the availability of monitoring data at runtime and compiler optimizations. If the developer wants access to certain data at runtime, then such a de-optimization may be acceptable. In fact, the extraction of such data from highly-optimized code may be extremely difficult and even misleading[7, 24] without these local de-optimizations.

3.1. Source code annotations

Monitoring assertions have two different forms: one to declare data and one to access declared data. An application first must declare which application-specific data is available for interactive monitoring; then, at various places throughout the code, the application allows access to that declared data via assertions.

Declarations. Declarations provide monitoring assertions with type and scoping information while limiting the amount of data items available to the interactor. Compiler support for declarations would reduce the instrumentation for this declaration statement to a simple tag on the language variable declaration (e.g., `monitor int iter;`). Figs. 5 illustrates a declaration sequence for a set of variables. As

in Fig.5, these declarations are currently preprocessor directives, `AAA_int(k)`, that expand into a procedure call with parameters for name, type, address, size, thread identifier, filename and line number:

```
aaa_decl_data("k", AAA_INT, &k,
  sizeof(k), tid, "slave1.c", 689)
```

The declaration for each `name` must occur prior to any access points using that name.

When this call is executed by the application with monitoring assertions, it stores this information. Later, this information can be written to an instrumentation signatures history file (see §4). The `aaa_decl_data` call first checks `name` to determine if it is already registered. If `name` does not exist, then it creates a table entry and initializes statistics for that name. If it does exist, then it verifies the instrumentation location (i.e., line number and file name). If `name` has been defined at another instrumentation location, `aaa_decl_data` indicates a name collision and does not register the offending name. If the static instrumentation location (sid) verifies, then it checks the thread identifier tid. If tid exists, then just the statistics are updated. Otherwise, monitoring assertions creates a new runtime instrumentation location (rid) and adds this information to the existing static instrumentation location in the registry.

Access points. Once data has been declared, monitoring assertions require control flow information about how the monitoring system can 'safely' access these data items at runtime. An assertion tells the monitoring system when the data is semantically meaningful as defined by the developer as well as when it is not being accessed by the current thread. These assertions are currently preprocessor directives that expand into a procedure call with parameters for thread identifier, filename, line number and a list of pairs of name and address. In Fig. 5, `AAA_Access(multi->err_multi, wu, iter, k, f, tolerance)` expands to

```
aaa_access(tid,"slave1.c",704,
  "multi->err_multi", &(multi->err_multi),
  "wu", &(wu), ... , NULL );
```

When this call is executed by the application with monitoring assertions, `aaa_access` acts like a declaration. The `aaa_access` call checks the static instrumentation location (sid) to determine if it is already registered. If sid does not exist, then it creates a table entry and initializes statistics for that location, else it checks the thread identifier tid as well as the data items in the list. If tid exists, then only the statistics are updated. Otherwise, the monitoring assertion creates a new runtime instrumentation location (rid) and adds it to the static instrumentation location in the registry. Essentially, an unlimited number of declared data items can be listed with each `aaa_access` call. At each appearance of a new sid, the system verifies the data items are previously declared using `name` to scan the declared data table. This provides the sid with type and size information.

4. Instrumentation signatures

Instrumentation signatures further augment monitoring assertions by providing a runtime history of an application's instrumentation. These signature files capture three basic characteristics: declarations, instrumentation points, and per-thread statistics about the number of hits on each instrumentation point (i.e., $Hits(sid, tid)$ and $Hits(sid)$). The monitoring system can use this information to determine how to bind mechanisms to access points so that performance is optimized. Signatures are generated when the target system is executed in *reference run* mode; signatures are then used at runtime to determine what data is available and to help predict which instrumentation points best satisfy data rates for the interactive requests. Instrumentation signatures do provide information unavailable with compile-time analysis and executable editing. In general, instrumentation signatures are a function of the application, its particular monitoring assertions, input data, and other system conditions such as the number of threads used by the application.

A *reference run* will produce only an initial guess about the instrumentation signatures value. For the relatively stable scientific applications considered in our research, this guess is reasonable and is not subject to frequent changes.

Declarations. The *declaration* section details an application component's name, location, type, size and its unique identifier. Declarations provide type and size information to monitoring assertions.

Instrumentation points. The *instrumentation points* section details all instrumentation points encountered during the reference run. Each line contains the location of the instrumentation point, a unique identifier and all the declared application components that are available at that instrumentation point. Using this knowledge, an interactive monitoring system can quickly determine what data is available and at what rates.

Statistics. The *statistics* section provides frequencies of execution of each instrumentation point including total hits $Hits(sid)$ as well as hits per thread $Hits(sid, tid)$—both raw and normalized statistics. Using this information, a monitoring system can choose instrumentation points based on their expected $Hits(sid)$.

5. Conclusions

We have introduced monitoring assertions and instrumentation signatures as two new techniques to help improve the efficiency and usability of monitoring systems. Monitoring assertions generalize application-specific instrumentation so that monitoring systems can dynamically control the binding of instrumentation points to monitoring mechanisms. Dynamic binding and the use of consistency information

can improve application perturbation and monitoring latencies. In addition, instrumentation signatures may be used to predict the frequencies and performance of instrumentation. Currently, a C library allows tools to dynamically bind mechanisms to these assertions.

Compiler integration of monitoring assertions could drastically alter their utility and efficiency. Application-specific instrumentation usually must supplement information gathered from other compiler-provided sources. If our monitoring assertions were integrated with a compiler, they could provide the following advantages: (1) use of compile-time type information to automate the instrumentation process; (2) integrate data dependency information with monitoring assertions to rearrange and combine assertions to improve performance; and, (3) use stack frame information to make monitoring assertions more precise.

References

[1] G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proc. ACM SIGPLAN Programming Language Design and Implementation (PLDI)*, 1997.

[2] T. Anderson and E. Lazowska. Quartz: A tool for tuning parallel program performance. In *Proc. 1990 SIGMETRICS Conf. Measurement and Modeling Computer Systems*, pages 115–125, Boston, 1990.

[3] T. Ball and J. Larus. Optimally profiling and tracing programs. *ACM Trans. Programming Languages and Systems*, 16(4):1319–60, 1994.

[4] P. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Trans. Computer Systems*, 13(1):1–31, 1995.

[5] D. Bhatt. Scalable parallel instrumentation (SPI): an environment for developing parallel system instrumentation. In *Proc. Intel Supercomputer Users Group Conf.*, pages 98–104, 1994.

[6] D. Brown, S. Hackstadt, A. Malony, B. Mohr, J. Dongarra, and B. Tourancheau. Program analysis environments for parallel language systems: the tau environment. In *Proc. Second Workshop on Environments and Tools for Parallel Scientific Computing*, pages 162–71, 1994.

[7] M. Copperman and C. McDowell. A further note on hennessy's 'symbolic debugging of optimized code'. *ACM Trans. Programming Languages and Systems*, 15(2):357–65, 1993.

[8] S. Graham, P. Kessler, and M. McKusick. Gprof: A call graph execution profiler. *SIGPLAN Notices (SIGPLAN '82 Symp. Compiler Construction)*, 17(6):120–126, 1982.

[9] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, and N. Mallavarupu. Falcon: On-line monitoring and steering of large-scale parallel programs. In *Proc. Frontiers of Massively Parallel Computation*, 1995.

[10] W. Gu, J. Vetter, and K. Schwan. An annotated bibliography of interactive program steering. *SIGPLAN Notices*, 29(9):140–8, 1994.

[11] S. Hackstadt, A. Malony, L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert. Distributed array query and visualization for high performance fortran. In *Proc. Euro-Par '96 Parallel Processing.*, pages 55–63, 1996.

[12] E. Kraemer and J. Stasko. The visualization of parallel systems: An overview. *Jour. Parallel and Distributed Computing*, 18(2):105–117, 1993.

[13] J. Kundu and J. Cuny. A scalable, visual interface for debugging with event-based behavioral abstraction. In *Proc. Fifth Symp. the Frontiers of Massively Parallel Computation*, pages 472–9, 1995.

[14] A. Malony. Event-based performance perturbation: a case study. *SIGPLAN Notices (Third ACM SIGPLAN Symp. Principles and Practice of Parallel Programming)*, 26(7):201–12, 1991.

[15] A. Malony, D. Reed, and H. Wijshoff. Performance measurement intrusion and perturbation analysis. *IEEE Trans. Parallel and Distributed Systems*, 3(4):433–50, 1992.

[16] D. Marinescu, H. Siegel, J. Lumpp, and T. Casavant. Models for monitoring and debugging tools for parallel and distributed software. *Jour. Parallel and Distributed Computing*, 9:171–184, 1990.

[17] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.

[18] B. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Trans. Parallel and Distributed Systems*, 1:206–217, 1990.

[19] D. Ogle, K. Schwan, and R. Snodgrass. Application-dependent dynamic monitoring of distributed and parallel systems. *IEEE Trans. Parallel and Distributed Systems*, 4(7):762–778, 1993.

[20] D. Reed, K. Shields, W. Scullin, L. Tavera, and C. Elford. Virtual reality and parallel systems performance analysis. *Computer*, 28(11):57–67, 1995.

[21] R. Ribler, J. Vetter, H. Simitci, and D. Reed. Autopilot: adaptive control of distributed applications. In *Proc. Seventh IEEE Int'l Symp. High Performance Distributed Computing*, 1998.

[22] J. Vetter and K. Schwan. High performance computational steering of physical simulations. In *Proc. Int'l Parallel Processing Symp.*, pages 128–132, Geneva, 1997.

[23] A. Waheed and D. Rover. A structured approach to instrumentation system development and evaluation. In *Proc. Supercomputing 95*, pages 1–1, 1995.

[24] R. Wismuller. Debugging of globally optimized programs using data flow analysis. *SIGPLAN Notices (ACM SIGPLAN '94 Conf. Programming Language Design and Implementation)*, 29(6):278–89, 1994.

[25] J. Yan, E. Hesham, and B. Shriver. Performance tuning with AIMS-an automated instrumentation and monitoring system for multicomputers. In *Proc. Twenty-Seventh Hawaii Int'l Conf. System Sciences. Vol.II: Software Technology*, pages 625–33, 1994.